TIAA: A Visual Toolkit for Intrusion Alert Analysis

Peng Ning, Pai Peng, Yiquan Hu, and Dingbang Xu

Department of Computer Science,
North Carolina State University, Raleigh, NC 27695-7534

Abstract. This paper presents the development of TIAA, a visual toolkit for intrusion alert analysis. TIAA is developed to provide an interactive platform for analyzing potentially large sets of intrusion alerts reported by heterogeneous intrusion detection systems (IDSs). To ensure timely response from the system, TIAA adapts main memory index structures and query optimization techniques to improve the efficiency of intrusion alert correlation. TIAA includes a number of useful utilities to help analyze potentially intensive intrusion alerts, including *alert aggregation/disaggregation*, *clustering analysis*, *focused analysis*, *frequency analysis*, *link analysis*, and *association analysis*. Moreover, TIAA provides several ways to visualize the analysis results, making it easier for a human analyst to understand the analysis results. It is envisaged that a human analyst and TIAA form a man-machine team, with TIAA performing automated tasks such as intrusion alert correlation and execution of analysis utilities, and the human analyst deciding what sets of alerts to analyze and how the analysis utilities are applied.

Key Words: Alert Correlation, Attack Scenario Analysis, Visualization

1 Introduction

Intrusion detection has been studied for more than 20 years. Various kinds of research and commercial intrusion detection systems (IDSs) have been developed and deployed. Examples of IDSs include EMERALD [1], STAT series scenario-based IDSs [2, 3], Snort [4], and ISS's RealSecure[1]. As a security mechanism complementary to intrusion prevention techniques such as access control and authentication, intrusion detection serves as a second line of defense of today's networks and information systems.

However, intrusion detection is still facing several challenges. Besides the inability to detect unknown attacks without generating a large number of false alerts, most of today's IDSs suffer from two additional problems. First, most of the IDSs focus on low-level, individual attacks or anomalies, without capturing the logical steps or strategies behind these attacks. As a result, human users have to figure out the connections between alerts. Second, current IDSs usually generate a large number of alerts, which include both real and false alerts. In situations where there are intensive intrusions, not only are actual alerts mixed with false alerts, but the number of alerts also becomes unmanageable. As a result, it is difficult for human users or intrusion response systems to understand the intrusions behind the alerts and take appropriate actions.

[1] http://www.iss.net.

In this paper, we present the development of TIAA, a visual toolkit for intrusion alert analysis, which is intended to address the aforementioned problems and thus to improve the usability of the current IDSs.

The primary goal of TIAA is to provide system support for interactive analysis of intrusion alerts reported by misuse detection systems. TIAA is based on the alert correlation techniques that we developed in [5] and [6]. Besides the previous results, in this paper, we develop several additional techniques to improve alert analysis. First, to ensure timely response from TIAA, we adapt several main memory index structures and query optimization techniques to improve the efficiency of intrusion alert correlation. Second, in addition to the analysis utilities we proposed in [6], we develop several other utilities to facilitate the analysis of potentially large sets of intrusion alerts, including *alert aggregation/disaggregation, clustering analysis, frequency analysis, link analysis*, and *association analysis*. Finally, we develop two additional visual representations of analysis results besides hyper-alert correlation graphs proposed in [5], making it easier for a human analyst to understand the analysis results. It is envisaged that a human analyst and TIAA form a man-machine team, with TIAA performing automated tasks such as intrusion alert correlation and execution of analysis utilities, and the human analyst deciding what sets of alerts to analyze and how the analysis utilities are applied.

The remainder of this paper is organized as follows. Section 2 presents the architecture of TIAA. Section 3 gives the efficient alert correlation algorithm, which adapts main memory index structures and query optimization techniques. Section 4 presents the interactive analysis utilities. Section 5 briefly describes the implementation of TIAA. Section 6 discusses related work, and section 7 concludes this paper and points out some future research directions.

2 TIAA Architecture

TIAA is composed of three subsystems, *alert collection subsystem, alert correlation subsystem*, and *interactive analysis subsystem*, which are centered around a *knowledge base* and a *database*. Figure 1 shows the architecture of TIAA.

The knowledge base stores the prior knowledge of various types of alerts possibly raised by IDS sensors. Such knowledge is represented in the form of a *hyper-alert type* for each type of alerts. A hyper-alert type consists of the alert attributes, the prerequisite, and the consequence of the alerts. Informally, the prerequisite of an alert is the necessary condition for the corresponding attack to succeed, and the consequence is the possible outcome of the attack. (Further details of alert correlation will be given in Section 3.) Both prerequisites and consequences are represented as predicates that take alert attributes as arguments. The knowledge base thus stores a dictionary of predicates, the implication relationships between these predicates, and a collection of hyper-alert types. Moreover, to facilitate the management of a potentially large number of intrusion alerts, TIAA stores all the alerts and the analysis results in the database.

The purposes of the alert collection subsystem are to resolve the potential naming inconsistency in different types of IDS sensors and to facilitate the central storage of intrusion alerts. Ideally, a common intrusion alert format such as IDMEF [7], if adopted, should revolve possible naming conflicts in intrusion alerts. However, in practice, possi-

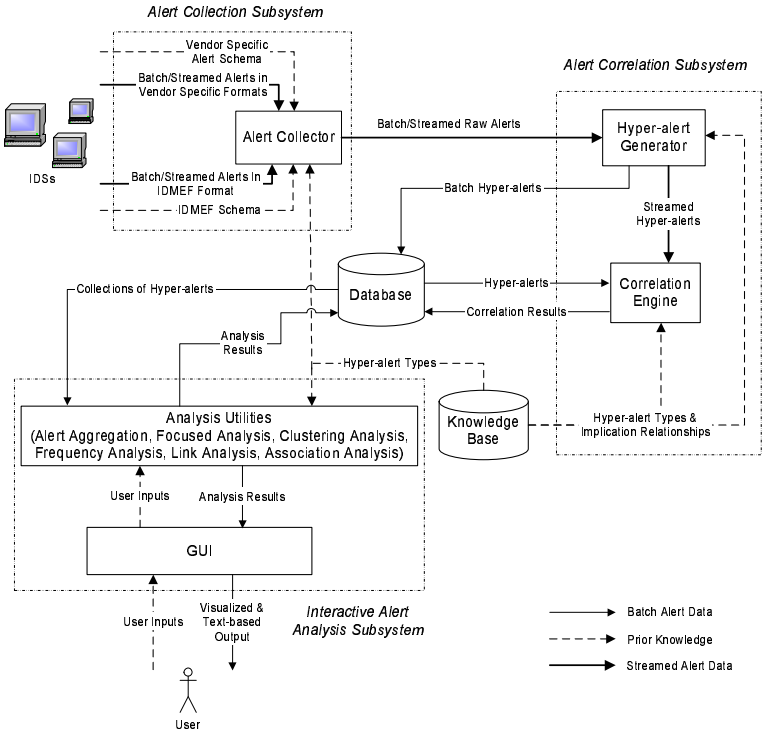

Fig. 1. The Architecture of TIAA

ble inconsistency still remains. For example, IDMEF allows vendor specific names for the classification of alerts, and it is possible to have different classifications in different IDSs for the same attack. The current version of TIAA supports IDMEF alerts and ISS's RealSecure. More vendor specific alerts can be supported by adding additional software components.

Alert correlation forms the foundation of intrusion alert analysis in TIAA. Our alert correlation is centered around *hyper-alerts*. Intuitively, a hyper-alert is an instance of a hyper-alert type, and may correspond to one or many intrusion alerts reported by the IDS sensors. The alert correlation subsystem is composed of two components: *hyper-alert generator* and *correlation engine*. The hyper-alert generator produces hyper-alerts based on the intrusion alerts collected by the alert collection subsystem, and at the same time, derive the prerequisites and the consequences of the hyper-alerts using the information stored in the knowledge base. The correlation engine then uses these results to perform the actual correlation.

The interactive alert analysis subsystem is the most critical part of TIAA; it consists of a set of interactive analysis utilities and a graphical user interface (GUI). The utilities are developed in such a way that they can be applied independently to a collection of hyper-alerts. As a result, these utilities can be iteratively applied to a collection of hyper-

alerts generated by previous analysis. The analysis results are visualized in a graphical form, and presented to the user through the GUI. These utilities allow roll-up/drill-down analysis of alerts. Intuitively, a human analyst may have high-level view of a potentially large set of intrusion alerts, and gradually investigate specific details as he/she knows more about the alerts.

There are currently six utilities in our system: alert aggregation/disaggregation, clustering analysis, focused analysis, frequency analysis, link analysis, and association analysis. The input of each utility is a collection of hyper-alerts (or the collection of raw alerts corresponding to these hyper-alerts) generated by the correlation engine or an analysis utility. The output is either another collection of hyper-alerts, or properties of the input hyper-alerts. The output results are saved back into the database for possible further process. The details of the interactive analysis model and the utilities will be discussed in Section 4.

3 Efficient Intrusion Alert Correlation

Intrusion alert correlation in TIAA is based on our techniques presented in [5], which have been shown to be promising in constructing high-level attack scenarios from low-level alerts.

Although the techniques in [5] provide a foundation for alert correlation in TIAA, a few challenges remain to be addressed. In particular, our previous implementation in [5] was aimed at validating the correlation techniques; little attention was paid to the efficiency issue. As a result, correlating a large set of alerts usually results in a long delay. Such performance is clearly not suitable for interactive analysis. Our timing analysis indicates that the main reason for this delay is that the previous implementation relies on the DBMS to perform the correlation, and the performance bottleneck lies in the interaction between the alert correlation program and the DBMS. To address this problem, we investigate a number of main memory index structures (*e.g.*, Linear Hashing [8] and T Trees [9]), and develop a few techniques to perform alert correlation in main memory by taking advantage of the unique features of intrusion alert correlation.

To be self-contained, in this section, we first give a brief overview of the intrusion alert correlation model (See [5] for more details.), and then discuss our efficient correlation techniques.

3.1 Overview of the Intrusion Alert Correlation Model

The alert correlation model is based on the observation that in series of attacks, the component attacks are usually not isolated, but related as different stages of the attacks, with the early ones preparing for the later ones. For example, an attacker has to install Distributed Denial of Service (DDOS) daemon programs before he can launch a DDOS attack. To take advantage of this observation, we correlate alerts using prerequisites and consequences of the corresponding attacks. Intuitively, the *prerequisite* of an attack is the necessary condition for the attack to be successful. For example, the existence of a vulnerable service is the prerequisite of a remote buffer overflow attack against the service. Moreover, an attacker may make progress (*e.g.*, install a Trojan horse program)

as a result of an attack. Informally, we call the possible outcome of an attack the *consequence* of the attack. In a series of attacks where earlier ones are launched to prepare for later ones, there are usually connections between the consequences of the earlier attacks and the prerequisites of the later ones. Accordingly, we identify the prerequisites (*e.g.*, existence of vulnerable services) and the consequences (*e.g.*, discovery of vulnerable services) of attacks and correlate detected attacks (*i.e.*, alerts) by matching the consequences of previous alerts and the prerequisites of later ones.

We use predicates as basic constructs to represent prerequisites and consequences of attacks. For example, a scanning attack may discover UDP services vulnerable to certain buffer overflow attacks. We can use the predicate *UDPVulnerableToBOF* (*VictimIP, VictimPort*) to represent this discovery. In general, we use a logical formula, *i.e.*, logical combination of predicates, to represent the prerequisite of an attack. Thus, we may have a prerequisite of the form *UDPVulnerableToBOF* (*VictimIP, VictimPort*) $\wedge$ *UDPAccessibleViaFirewall* (*VictimIP, VictimPort*). Similarly, we use a set of logical formulas to represent the consequence of an attack.

With predicates as basic constructs, we use a *hyper-alert type* to encode our knowledge about each type of attacks. A *hyper-alert type T* is a triple (*fact, prerequisite, consequence*) where (1) *fact* is a set of attribute names, each with an associated domain of values, (2) *prerequisite* is a logical formula whose free variables are all in *fact*, and (3) *consequence* is a set of logical formulas such that all the free variables in *consequence* are in *fact*. Intuitively, the *fact* component of a hyper-alert type gives the information associated with the alert, *prerequisite* specifies what must be true for the attack to be successful, and *consequence* describes what could be true if the attack indeed happens.

Given a hyper-alert type *T* = (*fact, prerequisite, consequence*), a *hyper-alert (instance) h of type T* is a finite set of tuples on *fact*, where each tuple is associated with an interval-based timestamp [*begin_time, end_time*]. The hyper-alert h implies that *prerequisite* must evaluate to True and all the logical formulas in *consequence* might evaluate to True for each of the tuples. The *fact* component of a hyper-alert type is essentially a relation schema (as in relational databases), and a hyper-alert is a relation instance of this schema. A hyper-alert *instantiates* its *prerequisite* and *consequence* by replacing the free variables in *prerequisite* and *consequence* with its specific values. Note that *prerequisite* and *consequence* can be instantiated multiple times if *fact* consists of multiple tuples. For example, if an IPSweep attack involves several IP addresses, the *prerequisite* and *consequence* of the corresponding hyper-alert type will be instantiated for each of these addresses.

To correlate hyper-alerts, we check if an earlier hyper-alert *contributes* to the prerequisite of a later one. Specifically, we decompose the prerequisite of a hyper-alert into parts of predicates and test whether the consequence of an earlier hyper-alert makes some parts of the prerequisite True (*i.e.*, makes the prerequisite easier to satisfy). If the result is positive, then we correlate the hyper-alerts. In our formal model, given an instance h of the hyper-alert type *T* = (*fact, prerequisite, consequence*), the *prerequisite set (or consequence set, resp.) of* h, denoted $P(h)$ (or $C(h)$, resp.), is the set of all such predicates that appear in *prerequisite* (or *consequence*, resp.) whose arguments are replaced with the corresponding attribute values of each tuple in h. Each element in $P(h)$ (or $C(h)$, resp.) is associated with the timestamp of the corresponding tuple in

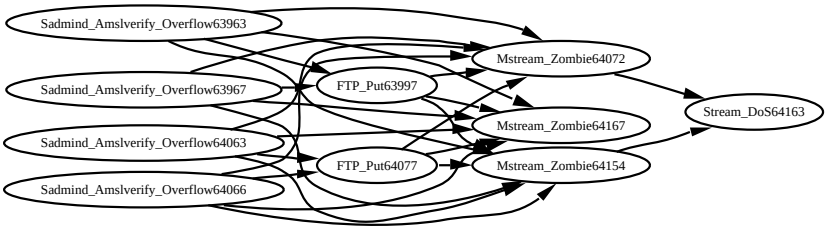

Fig. 2. A hyper-alert correlation graph discovered in our previous experiments

h. We say that hyper-alert h_1 *prepares for* hyper-alert h_2 if there exist $p \in P(h_2)$ and $C \subseteq C(h_1)$ such that for all $c \in C$, $c.end\_time < p.begin\_time$ and the conjunction of all the logical formulas in C implies p.

Given a sequence S of hyper-alerts, a hyper-alert h in S is a *correlated hyper-alert* if there exists another hyper-alert h' such that either h prepares for h' or h' prepares for h. We use a *hyper-alert correlation graph* to represent a set of correlated hyper-alerts. Specifically, a *hyper-alert correlation graph* $CG = (N, E)$ is a connected graph, where N is a set of hyper-alerts and for each pair $n_1, n_2 \in N$, there is a directed edge from n_1 to n_2 in E if and only if n_1 prepares for n_2. Figure 2 shows one of the hyper-alert correlation graphs discovered in our experiments with the 2000 DARPA intrusion detection evaluation datasets. Each node in Figure 2 represents a hyper-alert, where the label inside the node is the hyper-alert type followed by the hyper-alert ID.

3.2 Efficient Correlation Using Main Memory Query Optimization Techniques

In our previous work [5], we expanded the consequence set of each hyper-alert by including all predicates implied by the consequence set. We call the result the *expanded consequence set* of the hyper-alert. The predicates in both prerequisite and expanded consequence sets of the hyper-alerts are then encoded into strings called *Encoded Predicate* and stored in two tables, *PrereqSet* and *ExpandedConseqSet*, along with the corresponding hyper-alert ID and timestamp. Both tables have attributes *HyperAlertID*, *EncodedPredicate*, *begin_time*, and *end_time*, with meanings as indicated by their names. As a result, alert correlation can be performed using the following SQL statement[2].

```
SELECT DISTINCT c.HyperAlertID, p.HyperAlertID
FROM PrereqSet p, ExpandedConseqSet c
WHERE p.EncodedPredicate = c.EncodedPredicate AND c.end_time < p.begin_time
```

The essential problem of efficient alert correlation is to perform this SQL query efficiently. One option is to directly use database query optimization techniques, which

[2] This SQL statement is simplified for the sake of discussion. When alerts are reported from multiple IDS sensors, the sensor id and the maximum timestamp difference (which is the maximum difference between the timestamps of alerts raised by different IDS sensors for the same attack) must be taken into consideration.

have been studied extensively for both disk based and main memory databases. However, alert correlation has distinctive access patterns than typical database applications, and the unique characteristics in alert correlation present opportunities for further improvement. Thus, we decide to adapt existing query optimization techniques, especially main memory index structures such as T Trees [9] and Linear Hashing [8], by taking advantage of the unique features of alert correlation, seeking more performance gains than that can be achieved by generic query optimization techniques.

Correlating Intrusion Alerts (without Memory Constraint) Figure 3 presents a nested loop method that can accommodate streamed alerts. (As the name suggests, nested loop correlation is adapted from nested loop join [10].) It assumes that the input hyper-alerts are ordered ascendantly in terms of their beginning time. With such methods, an alert correlation system can be pipelined with IDS and produce correlation result in a timely manner.

The nested loop method takes advantage of main memory index structures. Many different kinds of index structures have been proposed in the literature. In our study, we focus on the following ones: Array Binary Search [11], AVL Trees [12], B Trees [13], Chained Bucket Hashing [14], Linear Hashing [8], and T Trees [9]. (Details of these index structures can be found in the corresponding references.)

While processing the hyper-alerts, the nested loop method maintains an index structure $\mathcal{I}$ for the instantiated predicates in the expanded consequence sets along with the corresponding hyper-alerts. Each time when a hyper-alert h is processed, the algorithm searches in $\mathcal{I}$ for each instantiated predicate p that appears in h's prerequisite set. A match of a hyper-alert h' implies that h' has the same instantiated predicate p in its expanded consequent set. If h'.EndTime is before h.BeginTime, then h' prepares for h. If the method processes all the hyper-alerts in the ascending order of their beginning time, it is easy to see that the nested loop method can find all and only the prepare-for relations between the input hyper-alerts.

We develop two adaptations to improve the performance of these index structures. Our first adaptation is based on the following observation.

Observation 1 *Multiple hyper-alerts may share the same instantiated predicate in their expanded consequence sets. Almost all of them prepare for a later hyper-alert that has the same instantiated predicate in its prerequisite set.*

Observation 1 implies that we can associate hyper-alerts with an instantiated predicate p if p appears in the expanded consequence sets of all these hyper-alerts. As a result, locating an instantiated predicate directly leads to the locations of all the hyper-alerts that share the instantiated predicate in their expanded consequence sets. We call the set of hyper-alerts associated with an instantiated predicate a *hyper-alert container*.

However, using hyper-alert containers does not always result in better performance. There are two types of accesses to the index structure in the nested loop correlation method (Figure 3): insertion and search. For the index structures that preserve the order of data items in them, insertion implies search, since each time when an element is inserted into the index structure, we have to place it in the "right" place. Using hyper-alert container does not increase the insertion cost significantly, while at the same time

Outline of Nested Loop Correlation
Input: A list H of hyper-alerts ordered ascendantly in their beginning times.
Output: All pairs of (h', h) such that both h and h' are in H and h' prepares for h.
Method:
Maintain an index structure $\mathcal{I}$ for instantiated predicates in the expanded consequence sets of hyper-alerts. Each instantiated predicate is associated with the corresponding hyper-alert. Initially, $\mathcal{I}$ is empty.
1. **for** each hyper-alert h in H (accessed in the given order)
2. **for** each instantiated predicate p in the prerequisite set of h
3. Search the set of hyper-alerts with index key p in $\mathcal{I}$. Let H' be the result.
4. **for** each h' in H'
5. **if** (h'.EndTime$<$ h.BeginTime) **then** output (h', h).
6. **for** each p in the expanded consequence set of h
7. Insert p along with h into $\mathcal{I}$.
end

Fig. 3. Outline of the nested loop alert correlation methods

reduces the search cost. However, for the non-order preserving index structures such as Linear Hashing, insertion does not involve search. Using hyper-alert containers would force to perform a search, since the hyper-alerts have to be put into the right container. In this case, hyper-alert container decreases the search cost but increases the insertion cost, and it is not straightforward to determine whether the overall cost is decreased or not. We resolve this issue through extensive experimental studies [15].

Observation 2 *There is a small, static, and finite set of predicates. Two instantiated predicates are the same only if they are instantiated from the same predicate.*

Observation 2 leads to a *two-level index structure*. Each instantiated predicate can be split into two parts, the predicate name and the arguments. The top-level index is built on the predicate names. Since we usually have a static and small set of predicate names, we use Chained Bucket Hashing for this purpose. Each element in the top-level index further points to a second-level index structure. The second-level index is built on the arguments of the instantiated predicates. When an instantiated predicate is inserted into a two-level index structure, we first locate the right hash bucket based on the predicate name, then locate the second-level index structure within the hash bucket (by scanning the bucket elements), and finally insert it into the second-level index structure using the arguments.

We expect the two-level index structure to improve the performance due to the following reasons. First, since the number of predicates is small and static, using Chained Bucket Hashing on predicate names is very efficient. In our experiments, the size of the hash table is set to the number of predicates, and it usually takes one or two accesses to locate the second-level index structure for a given predicate name. Second, the two-level index structure decomposes the entire index structure into smaller ones, and thus reduces the search time in the second-level index.

We have performed an extensive set of experiments, and the results are described in a related technical report [15]. Because of the space limit, we cannot include the detalied results in this paper. In summary, these experiments demonstrated that (1) hyper-alert containers improve the efficiency of order-preserving index structures, with which an insertion operation involves search, (2) two-level index improves the efficiency of all index structures, and (3) a two-level index structure combining Chained Bucket Hashing and Linear Hashing with hyper-alert container is the most efficient for streamed alerts.

Correlating Intrusion Alerts with Limited Memory The previous approaches to in-memory alert correlation have assumed that all index structures fit in memory during the alert correlation process. This may be true for analyzing intrusion alerts collected during several days or weeks; however, in typical operational scenarios, the IDSs produce intrusion alerts continuously and the memory of the alert correlation system will eventually be exhausted. A typical solution is to use a "sliding window" to focus on alerts that are close to each other; at any given point in time, only alerts after a previous time point are considered for correlation. This is equivalent to ignoring "correlated" alerts that are too far from each other.

We adopt a sliding window which can accommodate up to t intrusion alerts. The parameter t is determined by the amount of memory available to the intrusion alert correlation system. Since our goal is to optimize the intrusion alert correlation process, we do not discuss how to choose the appropriate value of t in this paper. Each time when a new intrusion alert is coming, we check if inserting this new alert will result in more than t alerts in the index structure. If yes, we remove the oldest alert from the index structure. In either case, we will perform the same correlation process as in Section 3.2. It is also possible to add multiple intrusion alerts in batch. In this case, multiple old alerts may be removed from the index structure.

Using a sliding window in our application essentially implies deleting old intrusion alerts when there are more than t alerts in memory. This problem seems trivial at the first glance, since all the data structures have known deletion algorithms. However, we soon realize that the index structures we use in the previous algorithm are in terms of instantiated predicates. To remove the oldest intrusion alerts, we need to locate and remove alerts in terms of their timestamps. Thus, the previous index structures cannot be used for deleting alerts.

To address this problem, we add a *secondary data structure* to facilitate locating the oldest intrusion alerts. Since the intrusion alerts are inserted as well as removed in terms of their time order, we use a queue (simulated with a circular buffer) for this purpose. Each newly inserted intrusion alert also has an entry added into this queue, which points to its location in the *primary index structure* built in terms of the instantiated predicates. When we need to remove the oldest intrusion alert, we simply dequeue an alert, find its location in the primary index structure, and delete it directly. Indeed, this is more efficient than the generic deletion method of the order preserving index structures (*e.g.*, AVL Trees), since deletion usually implies search in those index structures.

We have also performed a series of experiments to study the performance of alert correlation when there is limited memory [15]. Our experiments indicated that even if there is memory constraint, two-level index combining Chained Bucket Hashing and

Linear Hashing is still the most efficient index structure for streamed alerts. Thus, we use the nested loop correlation with the above two-level index in the current TIAA implementation.

4 TIAA Interactive Analysis Utilities

The most critical part of TIAA is its interactive analysis subsystem, which consists of a set of interactive analysis utilities and a GUI. In this section, we first give an overview of these utilities, and then describe them individually in detail.

4.1 Overview

Intrusion alerts are organized into *collections of hyper-alerts* in TIAA's interactive analysis subsystem. Each collection consists of a number of hyper-alerts, and each hyper-alert corresponds to one or more *raw intrusion alerts* reported by IDS sensors. Hyper-alerts in different collections may correspond to the same raw intrusion alerts; however, each raw intrusion alert corresponds to at most one hyper-alert within one collection, and each hyper-alert belongs to one and only one collection.

Each utility takes a collection of hyper-alerts as input. Depending on the output, these utilities can be divided into two classes: *hyper-alert generating utilities* and *feature extraction utilities*. Intuitively, a hyper-alert generating utility outputs a new collection of hyper-alerts, while a feature extracting utility only outputs the properties of the input collection of hyper-alerts.

We have developed a number of utilities, including *alert aggregation/disaggregation*, *clustering analysis*, *focused analysis*, *frequency analysis*, *link analysis*, and *association analysis*. The first three utilities are hyper-alert generating utilities, while the following three are feature extraction utilities. Among these utilities, focused analysis has been discussed in our previous work [6], and alert aggregation/disaggregation and clustering analysis are extended from the *adjustable graph reduction* and *graph decomposition* presented in [6], respectively.

TIAA provides both text-based and visualized representations of the analysis results. Each collection of hyper-alerts can be visualized in two ways. The first is the hyper-alert correlation graph discussed in Section 3.1. (See Figure 2 for an example.) Hyper-alert correlation graphs provide an intuitive representation of the causal relationships between hyper-alerts. Because a hyper-alert may correspond to many alerts reported by IDS sensors, hyper-alert correlation graphs can provide a concise view of a potentially large number of raw intrusion alerts. Nevertheless, certain information, especially the exact time information, is hidden in a hyper-alert correlation graph.

In addition to hyper-alert correlation graphs, TIAA also provides a *chronological alert correlation graph* to represent correlated alerts. In a chronological alert correlation graph, each node represents one raw alert reported by an IDS sensor. Nodes are arranged vertically in the order in which they are reported, with reference to a time line. Each directed edge from one node to another represents that the former prepares for the latter. Chronological alert correlation graphs are intended for close examination of a small set

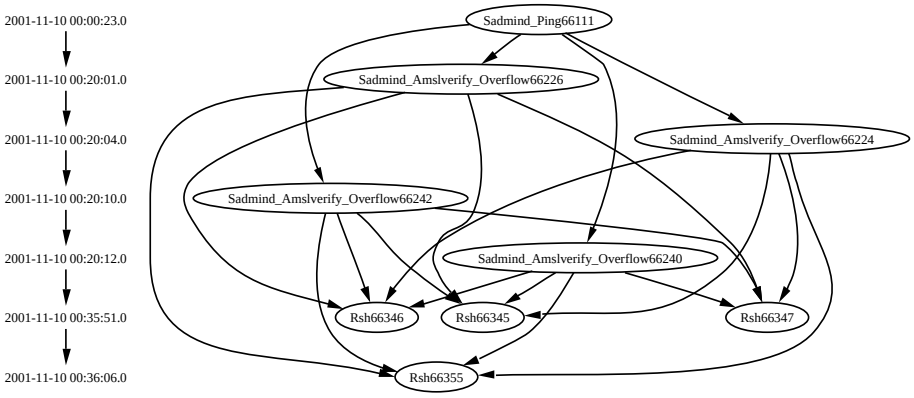

Fig. 4. A chronological alert correlation graph.

of correlated alerts; they could be difficult to read when there are a large number of alerts. Figure 4 shows an example of a chronological alert correlation graph.

TIAA also visualizes the results of link analysis. We will delay its discussion until the section of link analysis.

The GUI is a critical component to facilitate interactive analysis. With the support of the GUI, an analyst may examine various properties of intermediate analysis results. For example, an analyst may click on a node in a hyper-alert correlation graph to examine the attributes of all the raw alerts corresponding to this node. An analyst can then make appropriate decision about the next step in intrusion analysis.

Note that these utilities are intended for interactive analysis where it is usually difficult to determine what to do before the analyst sees the output of the previous step. We argue that this is often the case in practice. Thus, these utilities are developed as independent building blocks. It is certainly possible to automate some of them, if there is a clear reasoning framework as to how these utilities may be applied. Nevertheless, we focus on the description of individual utilities in this paper, while considering automated analysis process as our future work.

4.2 Alert Aggregation/Disaggregation

The purpose of alert aggregation is to reduce the complexity of a collection of hyper-alerts while keeping the structure of sequences of attacks reflected by these hyper-alerts. Alert disaggregation is to provide the flexibility to examine more detailed information about selected aggregated hyper-alerts.

The alert aggregation utility is extended from the adjustable graph reduction utility presented in [6], which is controlled by an *interval constraint*. Formally, given a time interval I (*e.g.*, 10 seconds), a hyper-alert h satisfies *interval constraint of* I if (1) h has only one tuple, or (2) for all t in h, there exist another t' in h such that there exist $t.begin\_time < T < t.end\_time$, $t'.begin\_time < T' < t'.end\_time$, and $|T - T'| < I$.

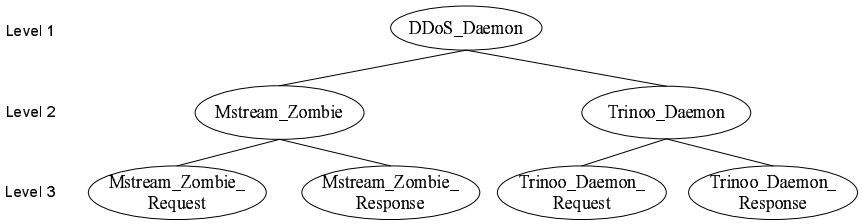

Fig. 5. An example abstraction hierarchy of hyper-alert types

Adjustable graph reduction allows hyper-alert aggregation only when the resulting hyper-alerts satisfy an interval constraint of a given threshold I. In other words, hyper-alerts can be aggregated only when they are close to each other. For example, in Figure 2, hyper-alerts 63963, 63967, 64063, and 64066 are all instances of hyper-alert type *Sadmind_Amslverify_Overflow*. Thus, we may aggregate them into one hyper-alert if permitted by the given interval constraint. Edges are reduced along with the aggregation of hyper-alerts. In Figure 2, the edges between the four *Sadmind_Amslverify_Overflow* hyper-alerts and the two *FTP_Put* hyper-alerts can be merged into a single edge. The larger a threshold I is, the more a hyper-alert correlation graph can be reduced. By adjusting the interval threshold, a user can control the degree to which a hyper-alert correlation graph is reduced.

Though adjustable graph reduction can simplify complex hyper-alert correlation graphs and thus improve their readability, one problem still remains. That is, there may be many types of alerts in a hyper-alert correlation graph. One incentive to have many types of alerts is to allow fine-grained names for different types of alerts and thus to keep more semantics along with the alerts. As a result, a reduced hyper-alert correlation graph may still have too many nodes and remain difficult to understand.

To allow further reduction of hyper-alert correlation graphs, we extend adjustable graph reduction by combining *abstraction* with interval constraints. Specifically, we generalize each type of hyper-alerts to a more general one. For example, RealSecure Network Sensor 7.0 may raise *Mstream_Zombie_Request* and *Mstream_Zombie_Response* alerts, which represent the request sent from an mstream master program to an mstream zombie program and the response, respectively. We may abstract both of them into one type of *Mstream_Zombie* alerts. Abstraction may be performed hierarchically so that there are different levels of abstractions. For example, we may further generalize *Mstream_Zombie* and *Trinoo_Daemon* into a type of *DDoS_Daemon* alert. We further assign an abstraction level to each (abstract) hyper-alert type to reflect the degree of abstraction. Figure 5 shows the abstraction hierarchy for the above examples.

Abstraction in alert aggregation is a dynamic process. An analyst may aggregate a set of hyper-alerts into one, and may also disaggregate an aggregated hyper-alert into several hyper-alerts. Both of them can be done in terms of the abstraction hierarchy. One way to effectively use alert aggregation/disaggregation is to uses the highest abstraction level for all hyper-alerts when performing alert aggregation for the first time. This ensures the most concise hyper-alert correlation graphs for a given interval constraint.

After getting the high-level idea of the alerts in the hyper-alert correlation graphs, the analyst may select hyper-alerts in the graph and disaggregate them by reducing their abstraction levels. TIAA can then regenerate the hyper-alert correlation graphs in a finer granularity for selected hyper-alerts. As a result, different levels of abstractions can be used for different hyper-alerts in the same hyper-alert correlation graph, and abstraction levels assigned to hyper-alerts have less impact to the analysis results.

4.3 Clustering Analysis

Intuitively, clustering analysis is to partition a collection of hyper-alerts into different groups so that the hyper-alerts in each group share certain common features. Clustering analysis is based on the *graph decomposition* utility in [6]. The difference is that clustering analysis in TIAA can be applied to any collection of hyper-alerts, while graph decomposition in [6] is limited to analyzing hyper-alert correlation graphs. To be self-contained, we briefly discuss clustering analysis here, despite the similarity between clustering analysis and graph decomposition.

Clustering analysis uses *clustering constraints* to specify the potential "common features" shared by hyper-alerts. Given two sets of attribute names A_1 and A_2, a *clustering constraint* $C_c(A_1, A_2)$ is a logical combination of comparisons between constants and attribute names in A_1 and A_2. A clustering constraint is a constraint for two hyper-alerts; the attribute sets A_1 and A_2 identify the attributes from the two hyper-alerts. For example, we may have two sets of attribute names $A_1 = \{SrcIP, DestIP\}$ and $A_2 = \{SrcIP, DestIP\}$, and $C_c(A_1, A_2) = (A_1.SrcIP = A_2.SrcIP) \wedge (A_1.DestIP = A_2.DestIP)$. Intuitively, this is to say that two hyper-alerts remain in the same cluster if they have the same source and destination IP addresses.

A clustering constraint $C_c(A_1, A_2)$ is *enforceable w.r.t. hyper-alert types* T_1 *and* T_2 if when we represent $C_c(A_1, A_2)$ in a disjunctive normal form, at least for one disjunct C_{ci}, all the attribute names in A_1 appear in T_1 and all the attribute names in A_2 appear in T_2. For example, the above clustering constraint is enforceable w.r.t. T_1 and T_2 if both of them have $SrcIP$ and $DestIP$ in the $fact$ component. Intuitively, a clustering constraint is enforceable w.r.t. T if it can be evaluated using two hyper-alerts of types T_1 and T_2, respectively.

If a clustering constraint $C_c(A_1, A_2)$ is enforceable w.r.t. T_1 and T_2, we can *evaluate* it with two hyper-alerts h_1 and h_2 that are of type T_1 and T_2, respectively. A clustering constraint $C_c(A_1, A_2)$ evaluates to True for h_1 and h_2 if there exists a tuple $t_1 \in h_1$ and $t_2 \in h_2$ such that $C_c(A_1, A_2)$ is True with the attribute names in A_1 and A_2 replaced with the values of the corresponding attributes of t_1 and t_2, respectively; otherwise, $C_c(A_1, A_2)$ evaluates to False. For brevity, we write $C_c(h_1, h_2)$ = True if $C_c(A_1, A_2)$ = True for h_1 and h_2. Then two hyper-alerts h_1 and h_2 are in the same cluster if $C_c(h_1, h_2)$ = True or $C_c(h_2, h_1)$ = True.

One application of clustering analysis is to decompose a hyper-alert correlation graph into smaller ones with the relationship between (the attributes of) hyper-alerts. This is the graph decomposition discussed earlier. (We have demonstrated its usefulness in our previous work [6].) In TIAA, clustering analysis has a more general application than graph decomposition, though the latter still remains an important utility as a special case of clustering analysis.

4.4 Focused Analysis

Focused analysis was first introduced in [6]. We include a brief discussion here to be self-contained.

Intuitively, focused analysis is to concentrate on selected alerts by filtering out those that do not satisfy a given focusing constraint. A *focusing constraint* is a logical combination of comparisons between attribute names and constants. For example, we may have a focusing constraint $SrcIP = 129.174.142.2 \vee DestIP = 129.174.142.2$. A focusing constraint C_f is *enforceable w.r.t. a hyper-alert type* T if when we represent C_f in a disjunctive normal form, at least for one disjunct C_{fi}, all the attribute names in C_{fi} appear in T. For example, the above focusing constraint is enforceable w.r.t. $T = (\{SrcIP, SrcPort\}, NULL, \emptyset)$, but not w.r.t. $T' = (\{IP, Port\}, NULL, \emptyset)$. Intuitively, a focusing constraint is enforceable w.r.t. T if it can be evaluated using a hyper-alert instance of type T.

We may *evaluate* a focusing constraint C_f with a hyper-alert h if C_f is enforceable w.r.t. the type of h. For a tuple t in h (which usually corresponds to a raw intrusion alert reported by an IDS sensor), we say a focusing constraint C_f *evaluates to True (or False) for* t if C_f is True (or False) with the attribute names replaced with the corresponding attribute values of t. For example, consider the aforementioned focusing constraint C_f, which is $SrcIP = 129.174.142.2 \vee DestIP = 129.174.142.2$, and a hyper-alert $h = \{(SrcIP = 129.174.142.2, SrcPort = 80), (SrcIP = 152.14.51.14, SrcPort = 1033)\}$, we can easily have that C_f = True for the first tuple in h, while C_f = False for the second one. We say a focusing constraint C_f evaluates to True for h if there exists at least one tuple $t \in h$ such that C_f evaluates to True for t; otherwise, C_f evaluates to False. The output of a focused analysis of a collection of hyper-alerts is the hyper-alerts with (only) the tuples for which the focusing constraint evaluates to True. In the above example, $h' = \{(SrcIP = 129.174.142.2, SrcPort = 80)\}$ remains in the output.

Focused analysis is particularly useful when we have certain knowledge of the alerts, the systems being protected, or the attacking computers. We expect an analyst to discover, or hypothesize and then verify, such knowledge while using the other utilities.

4.5 Frequency Analysis

Frequency analysis is developed to help an analyst identify patterns in a collection of alerts by counting the number of alerts that share some common features. Similar to clustering analysis, frequency analysis partitions the input collection of hyper-alerts. For example, an analyst may count the number of raw alerts that share the same destination IP address to find the most frequently hit target. For convenience, we reuse the notion of clustering constraints to specify the clusters.

Frequency analysis in TIAA can be applied in both *count mode* and *weighted analysis mode*. In count mode, frequency analysis simply counts the number of raw intrusion alerts that fall into the same cluster, while in weighted analysis mode, it adds all the values of a given numerical attribute (called weight attribute) of all the alerts in the same cluster. As an example of frequency analysis in weighted analysis mode, an analyst may use the priority of an alert type as the weight attribute, and learn the weighted frequency of alerts for all destination IP addresses.

For convenience reasons, frequency analysis automatically ranks the clusters ascendantly or descendantly in terms of the results. A filter which specifies a range of frequency values may be applied optionally so that only results that fall into this range are returned to the analyst.

The frequency analysis utility is conceptually equivalent to applying clustering analysis followed by a simple counting or summing for each of the clusters. However, since frequency analysis is developed for interactive analysis, it is much more convenient for an analyst if the utility combines all the operations together, especially when not all the clusters need to be reported to the analyst.

4.6 Link Analysis

Link analysis is intended to analyze the connection between entities represented by categorical attribute values. Examples include how two IP addresses are related to each other in a collection of alerts, and how IP addresses are connected to the alert types. Though link analysis takes a collection of hyper-alerts as input, it indeed analyzes the raw intrusion alerts corresponding to these hyper-alerts. Link analysis can identify candidate attribute values, evaluate their importance according to user defined metric, and rank them accordingly.

Link analysis takes at least two categorical attributes, A_1 and A_2 (*e.g.*, source IP and destination IP), as parameters. Similar to frequency analysis, link analysis may be used in *count mode* or *weighted analysis mode*. In the latter case, link analysis needs an additional weight attribute with a numerical domain. For each pair of attribute values $(A_1 = a_1, A_2 = a_2)$, link analysis with categorical attributes A_1 and A_2 counts the number of all the alerts that have $A_1 = a_1$ and $A_2 = a_2$ in count mode, or summarize the weight attribute values of these alerts in weighted analysis mode.

Given a link analysis with categorical attributes A_1 and A_2 over a collection of hyper-alerts, or equivalently, the corresponding set of raw intrusion alerts, we call each pair of attribute values a *link involving attributes* A_1 *and* A_2, denoted $(A_1 = a_1, A_2 = a_2)$. We then define the *weight of a link* $(A_1 = a_1, A_2 = a_2)$ as the number of alerts that have $A_1 = a_1$ and $A_2 = a_2$ in count mode, or the sum of the corresponding weight attribute values in weighted analysis mode. The *weight of an attribute value* is then the sum of the weights of the links involving the value. Specifically, the weight of $A_1 = a_1$ is the sum of the weights of all links that have a_1 as the value of A_1, while the weight of $A_2 = a_2$ is the sum of the weights of all links that have a_2 as the value of A_2.

Link analysis has two variations, *dual-domain link analysis* and *uni-domain link analysis*, depending on the treatment of the values of the two categorical attributes. In dual-domain link analysis, values of the two categorical alert attributes are considered different entities, even though they may have the same value. For example, we may perform a dual-domain link analysis involving source IP address and destination IP address. An IP address representing source IP is considered as a different entity from the same IP address representing a destination IP address. In contrast, uni-domain link analysis requires that the two attributes involved in link analysis must have the same domain, and the same value is considered to represent the same entity, no matter which attribute it corresponds to. In the earlier example, the same IP address represents the same host, whether it is a source or a destination IP address.

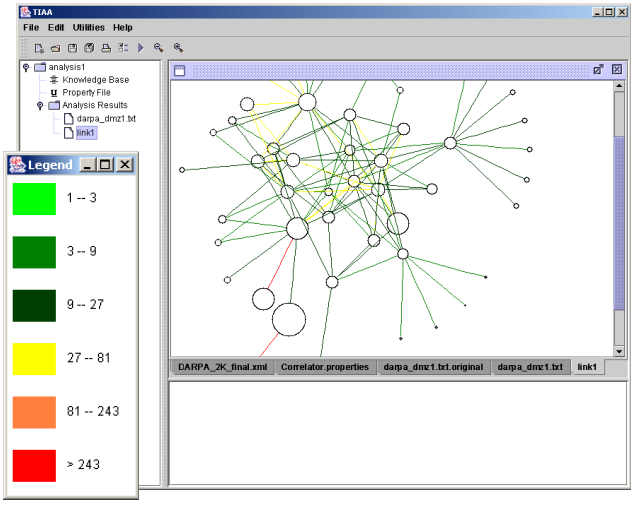

Fig. 6. Visual representation of a uni-domain link analysis (Note that edges are in different colors)

In TIAA, the result of a link analysis is visualized in a graphical format. Attribute values are represented as nodes in the (undirected) graph, with different sizes representing the weight of the corresponding attribute values. When uni-domain link analysis is used, all the nodes have the same shape (*e.g.*, circle); when dual-domain link analysis is used, two different shapes (*e.g.*, circle and square) correspond to the two different attributes, respectively. The link between two attribute values are represented by the edge connecting the corresponding nodes. The weight of each link is indicated by the color of the edge. Figure 6 shows an example of a uni-domain link analysis. Note that additional information (*e.g.*, attribute values) about each node or link can be obtained through the GUI.

Link analysis can be considered a special case of association analysis, which is discussed next. However, due to its simplicity and the visual representation of its results, we use link analysis as a separate utility in TIAA.

4.7 Association Analysis

Association analysis is used to find out frequent co-occurrences of values belonging to different attributes that represent various entities. For example, we may find through association analysis that many attacks are from source IP address 152.14.51.14 to destination IP address 129.14.1.31 at destination port 80. Such patterns cannot be easily found by frequency analysis, because before knowing the "pattern" of patterns, it is difficult to know how frequency analysis should be performed.

Association analysis is inspired by the notion of an association rule, which was first introduced in [16]. Given a set I of items, an *association rule* is a rule of the form $X \rightarrow Y$, where X and Y are subsets (called item sets) of I and $X \cap Y = \emptyset$. Association

rules are usually discovered from a set T of transactions, where each transaction is a subset of I. The rule $X \rightarrow Y$ has a support s in the transaction set T if $s\%$ of the transactions in T contain $X \cup Y$, and it has a confidence c if $c\%$ of the transactions in T that contain X also contain Y.

TIAA doesn't use association rules directly; instead, it uses the item sets that have large enough support (called *large item sets*) to represent the patterns embedded in alert attributes. We consider each alert a transaction. The large item sets discovered from the intrusion alerts then represent frequent attribute patterns in the alert set.

Syntactically, association analysis takes a set S of categorical alert attributes and a threshold t as parameters. Similar to frequency analysis and link analysis, association analysis can be applied in both count mode and weighted analysis mode. In the latter mode, association analysis requires a numerical attribute (also called a weight attribute) as an additional parameter. To facilitate the weighted analysis mode, we extend the notion of support to *weighted support*. Given a set X of attribute values and a weight attribute w, the *weighted support of X w.r.t. w* in the transaction set T is

$$\text{weighted support}_w(X) = \frac{\text{sum of } w \text{ of all alerts in } T \text{ that contain } X}{\text{sum of } w \text{ of all alerts in } T}.$$

Thus, association analysis of a collection of alerts in count mode finds all sets of attribute values that have support more than t, while association analysis of a collection of alerts in weighted analysis mode returns all sets of attribute values that have weighted support more than t.

5 Implementation

TIAA is implemented in Java, with JDBC to access the database. To save development effort, TIAA uses the GraphViz package[3] as the visualization engine to generate the graphical representation of the analysis results. TIAA relies on a knowledge base for prior knowledge about different types of alerts as well as implication relationships between predicates. Because of the need for human analysts to write and possibly revise the knowledge base, the knowledge base is represented in an XML format. TIAA uses the Apache Xerces2 Java Parser[4] to facilitate the manipulation of the knowledge base.

Figure 7 shows a screen shot of TIAA. The collections of hyper-alerts and analysis results are organized in a tree structure rooted at "Analysis Results" in the left window. Each node either represents a collection of hyper-alerts, or the result of applying an analysis utility to the collection of alerts represented by the corresponding parent node. Nodes representing analysis results are all terminal nodes. The analysis history is kept in the database. For each node, the utility and the parameters used to obtain the corresponding collection of hyper-alerts or analysis results are all stored in the database, and can be retrieved through the GUI. The top right window is the main output window, which displays all the visualized analysis results. These visualized results are further organized in tabs, which can be activated by clicking on the corresponding labels. The

[3] http://www.research.att.com/sw/tools/graphviz/

[4] http://xml.apache.org/xerces2-j/index.html.

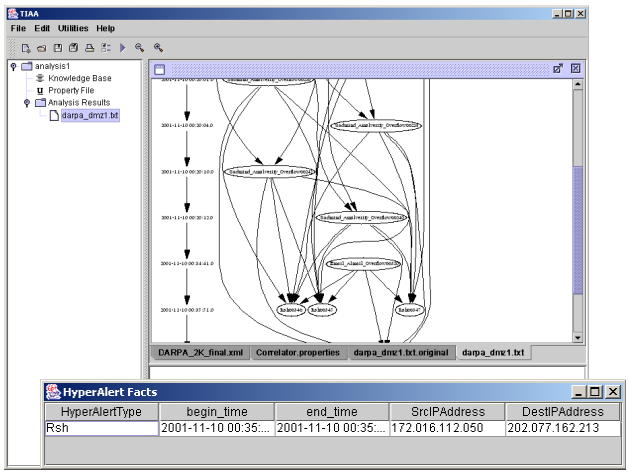

Fig. 7. A screen shot of TIAA

lower right window is used to output text-based information, such as the parameters of the analysis utilities. Additional floating windows can be activated to display requested information. For example, Figure 7 includes a floating window that displays the attribute values of a selected hyper-alert in the main output window.

6 Related Work

Research on intrusion alert correlation has been rather active recently. Early approaches (*e.g.*, Spice [17], probabilistic alert correlation [18]) correlate alerts based on the similarities between alert attributes. Though they are effective for correlating some alerts (*e.g.*, alerts with the same source and destination IP addresses), they cannot fully discover the causal relationships between related alerts.

Another class of methods (*e.g.*, correlation based on STATL [19] or LAMBDA [20], the Tivoli approach [21], and the data mining approach [22]) bases alert correlation on attack scenarios completely or partially specified by human users, or learned through training datasets. A limitation of these methods is that they are restricted to *known* attack scenarios.

A third class of method (*e.g.*, JIGSAW [23], the MIRADOR approach [24], and our previous approach [5, 6]) targets at recognizing multi-stage attacks; it correlates alerts if the precondition of some later alerts are satisfied by the consequences of some earlier alerts. Compared with the previous two classes of methods, this class can potentially uncover the causal relationship between alerts, and is not restricted to known attack scenarios. The results presented in this paper is continuance of the results in [5, 6]. Based on the previous results, we have developed methods to perform intrusion alert correlation efficiently, as well as new analysis utilities and visualization techniques to facilitate alert analysis.

A formal model named M2D2 was proposed in [25] to correlate alerts by using multiple information sources, including the characteristics of the monitored systems, the vulnerability information, the information about the monitoring tools, and information of the observed events. Due to the multiple information sources used in alert correlation, this method can potentially lead to better results than those simply looking at intrusion alerts.

A mission-impact-based approach was proposed in [26] to correlate alerts raised by INFOSEC devices such as IDS and firewalls. A distinguishing feature of this approach is that it correlates the alerts with the importance of system assets so that attention can be focused on critical resources.

Association rules have been used in network intrusion detection [27] as well as IDS sensor profiling [28]. However, association rules are used to automatically build misuse detection models in [27], and model the normal IDS sensor behaviors with bursts of alerts in [28], while in TIAA, association rules are used to discover patterns in intrusion alert attribute values.

7 Conclusions and Future Work

This paper presented the development of TIAA, a visual toolkit for intrusion alert analysis. TIAA is developed to provide an interactive platform for analyzing potentially large sets of intrusion alerts reported by heterogeneous intrusion detection systems (IDSs). It is envisaged that a human analyst and TIAA form a man-machine team, with TIAA performing automated tasks such as intrusion alert correlation and applications of analysis utilities, and the human analyst deciding what sets of alerts to analyze and how the analysis utilities are applied (through user input parameters).

Our future work is two-fold. First, we would like to develop more utilities to facilitate intrusion alert analysis. Second, we would like to gain more insights into the analysis process and thus automate as many analysis tasks as possible.

References

1. Porras, P.A., Neumann, P.G.: EMERALD: Event monitoring enabling response to anomalous live disturbances. In: Proceedings of the 20th National Information Systems Security Conference, National Institute of Standards and Technology (1997)
2. Vigna, G., Kemmerer, R.A.: NetSTAT: A network-based intrusion detection system. Journal of Computer Security **7** (1999) 37–71
3. Vigna, G., Kermmerer, R.A.: Designing a web of highly-configurable intrusion detection sensors. In: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001). (2001) 69–84
4. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 1999 USENIX LISA conference. (1999)
5. Ning, P., Cui, Y., Reeves, D.S.: Constructing attack scenarios through correlation of intrusion alerts. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, D.C. (2002) 245–254
6. Ning, P., Cui, Y., Reeves, D.S.: Analyzing intensive intrusion alerts via correlation. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), Zurich, Switzerland (2002) 74–94

7. Curry, D., Debar, H.: Intrusion detection message exchange format data model and extensible markup language (xml) document type definition. Internet Draft, draft-ietf-idwg-idmef-xml-03.txt (2001)
8. Litwin, W.: Linear hashing: A new tool for file and table addressing. In: Proceedings of the 6th Conference on Very Large Data Bases, Montreal, Canada (1980) 212–223
9. Lehman, T.J., Carey, M.J.: A study of index structure for main memory database management systems. In: Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan (1986) 294–303
10. Garcia-Molina, H., J. D. Ullman, J.W.: Database System Implementation. Prentice Hall (2000)
11. Ammann, A., Hanrahan, M., Krishnamurthy, R.: Design of a memory resident DBMS. In: Proceedings of IEEE COMPCON, San Francisco (1985)
12. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
13. Comer, D.: The ubiquitous B-Tree. ACM Computeing Surveys **11** (1979) 121–137
14. Knuth, D.: The Art of Computer Programming. Addison-Wesley (1973)
15. Ning, P., Xu, D.: Adapting query optimization techniques for efficient intrusion alert correlation. Technical Report TR-2002-14, North Carolina State University, Department of Computer Science (2002)
16. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Proc. of the 1993 Int'l Conf. on Management of Data. (1993) 207–216
17. Staniford, S., Hoagland, J., McAlerney, J.: Practical automated detection of stealthy portscans. Journal of Computer Security **10** (2002) 105–136
18. Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001). (2001) 54–68
19. Eckmann, S., Vigna, G., Kemmerer, R.: STATL: An Attack Language for State-based Intrusion Detection. Journal of Computer Security **10** (2002) 71–104
20. Cuppens, F., Ortalo, R.: LAMBDA: A language to model a database for detection of attacks. In: Proc. of Recent Advances in Intrusion Detection (RAID 2000). (2000) 197–216
21. Debar, H., Wespi, A.: Aggregation and correlation of intrusion-detection alerts. In: Recent Advances in Intrusion Detection. LNCS 2212 (2001) 85 – 103
22. Dain, O., Cunningham, R.: Fusing a heterogeneous alert stream into scenarios. In: Proceedings of the 2001 ACM Workshop on Data Mining for Security Applications. (2001) 1–13
23. Templeton, S., Levit, K.: A requires/provides model for computer attacks. In: Proceedings of New Security Paradigms Workshop, ACM Press (2000) 31 – 38
24. Cuppens, F., Miege, A.: Alert correlation in a cooperative intrusion detection framework. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy. (2002)
25. Morin, B., Mé, L., Debar, H., Ducassé, M.: M2D2: A formal data model for IDS alert correlation. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002). (2002) 115–137
26. Porras, P., Fong, M., Valdes, A.: A mission-impact-based approach to INFOSEC alarm correlation. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002). (2002) 95–114
27. Lee, W., Stolfo, S.: A framework for constructing features and models for intrusion detection systems. ACM Transactions on Information and System Security **3** (2000) 227–261
28. Manganaris, S., Christensen, M., Zerkle, D., Hermiz, K.: A data mining analysis of RTID alarms. Computer Networks **34** (2000) 571–577